SPECIFYING A VISUAL FILE SYSTEM IN Z

John Hughes, Department of Computing Science, University of Glasgow.

1. Introduction

We specify part of the system software of a well-known range of personal computers with a direct manipulation user interface. We refer to this software as product M, and the computers as product A. Product M provides a visual interface to a hierarchical file system. Files, discs, and folders (directories) are visible as icons on the screen. They may be moved or copied from place to place by dragging their icons around with the mouse. Dragging an icon into the "trash can" discards it. Discs and folders may be "opened", creating a window on the screen in which their contents can be viewed. We will specify these aspects of product M, using the Z specification language, and the specification style developed at Oxford.

A system is specified in Z by describing its *states* and the way *operations* affect those states. Both states and operations are specified using *schemas* — collections of variables related by predicates. Schemas are represented by named boxes, divided into an upper part containing variable declarations and a lower part containing predicates involving those variables. Variables after an operation are conventionally given dashed names. To reduce repetition schemas may be used as macros in other schemas, with dashed schemas representing consistent dashing of variables, and ΔS representing both S and S'. If macro expansion gives rise to duplicated declarations, they are considered equivalent to a single declaration.

The specification begins with a very abstract description of product M, to which more detail is added step by step. The Z schema calculus is used to build more detailed specifications from simpler ones, and to combine several detailed views to make the complete specification. This allows the specification of a relatively complex system to be built up from several simple parts, each specifying one aspect of the overall system.

Z is described in [Spivey] and [Hayes], which also contains many case studies of its application. The specification technique used in this paper was inspired by Sufrin's specification of an electronic mail system [Sufrin].

2. Product M's State: A Set of Objects

We focus our attention on the *objects* that the user manipulates directly — icons, possibly with associated windows. Let us introduce a set of all possible objects, containing at least the desktop and the trash can.

```
OBJ
desktop ∈ OBJ
trash ∈ OBJ
```

Every object has a *location*, which is another object. For example, a file might be located in a folder, on a disc, or on the desktop. Locations form a hierarchy. We specify the state as:

```
M1
| objects : ℙOBJ
| loc : OBJ -+> OBJ
|-------------
| desktop ∈ objects
| trash ∈ objects
| dom loc = objects - {desktop}
| rng loc  objects
| loc trash = desktop
| (loc⁻¹)* {{desktop}} = objects
```

Since $(loc^{-1})^*$ relates each object to everything below it in the hierarchy, $(loc^{-1})^*\ \{\{desktop\}\}$ is the set of objects reachable from the desktop. So the last line requires that every object is below the desktop in the hierarchy. Coupled with the fact that `loc` is a function, this condition implies that there are no cycles.

2.1 Movement

Now that we can talk about locations, we can begin to specify movement.

```
MOVE1
| ΔM1
| obj? : OBJ                    the object to be moved
| to? : OBJ                     the destination
|-------------
| obj? ∉ {desktop, trash}
| to? ∈ (loc⁻¹)* {{obj?}}
| loc' = loc ⊕ {obj? |-> to?}
```

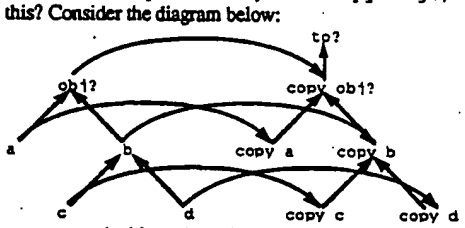
(By convention, the inputs to an operation are given names ending in a question mark.) The third condition above

specifies how the locations of objects are changed, while the second states a precondition that no cycle is created.

2.2 Copying

Product M also provides a way to make a new *copy* of an object, at a new location. The entire hierarchy below the copied object is also copied, so a single operation may create arbitrarily many new objects. We will need to refer to "the copy" of an object, so we introduce a one-to-one function `copy : OBJ >+> OBJ` mapping copied objects to their newly created copies. This association forms part of our description of the `COPY` operation, so appears in its schema.

The trickiest thing to specify is the copied hierarchy below `copy obj?`, with the same shape as that below `obj?`. How can we express this? Consider the diagram below:



The left side of this diagram represents the hierarchy being copied, the right hand side the new hierarchy, and the curved arrows the `copy` function. But we can pass from `copy d` (say) to its location by: first using $\texttt{copy}^{-1}$ to find `d` (recall `copy` is one-to-one!), finding the location of `d` (which is `b`), and then using `copy` to find the copy of `b`. So the new hierarchy on the right can be expressed as $\texttt{copy}^{-1} \fatsemi \texttt{loc} \fatsemi \texttt{copy}$ (where $\fatsemi$ denotes function composition).

COPY1

$$\Delta M1$$
$$obj?, to? : OBJ$$
$$copy : OBJ >+> OBJ$$

$$obj? \notin \{desktop, trash\}$$
$$dom\ copy = (loc^{-1})^* (\{obj?\})$$
$$rng\ copy \cap objects = \emptyset$$
$$loc' = loc \cup \{copy\ obj? \mapsto to?\} \cup copy^{-1} \fatsemi loc \fatsemi copy$$

3. Open and Closed Objects

Now let us begin to add more detail, and talk about *open* objects — those having an associated window on the screen displaying their contents, and *visible* objects. We regard an object as visible if it appears in any open window on the screen, including the desktop. (In practice, windows may overlap, and so it may be necessary to rearrange the windows on the screen to see some "visible" objects.)

M2

$$M1$$
$$open, visible : \mathbb{P} OBJ$$

$$open \subseteq objects$$
$$desktop \in open$$
$$visible = loc^{-1}(open)$$

It is easy to specify operations to open and close objects.

3.1 Movement

Now let us give a more detailed specification of the `MOVE` operation.

MOVE2

$$\Delta M2$$
$$MOVE1$$

$$obj? \in visible$$
$$to? \in visible \cup open$$
$$to? \neq trash \Rightarrow open' = open$$
$$to? = trash \Rightarrow open' = open \setminus \{obj?\}$$

The set of open objects is unchanged, *except* when an open object is moved into the trash can. It is then automatically closed. This special behaviour of the trash can seems rather anomolous. There is little wrong with treating the trash can exactly like every other object, and requiring simply `open' = open`. Reasonable alternatives might be to require objects in the trash can to be closed at all times, by modifying the `M2` invariant, or to close every object newly below the trash can in the hierarchy, allowing objects to be re-opened once in the trash can. It is hard to understand the rationale for the design actually implemented.

3.2 Copying

In the same way we can add detail to our specification of copying. The open objects change as follows: existing objects remain open if they were open before the operation, while copies of open objects are created open. The set of newly open objects is therefore the image of the set of open objects through the copy function.

```
COPY2 ──────────────────────────────
|       ΔM2
|       COPY1
|-------------------
|       obj? ∈ visible
|       to? ∈ visible ∪ open
|       open' = open ∪ copy (open)
──────────────────────────────────
```

4. Residence on Discs

Our second extension specifies explicitly on which disc objects reside, and how residence on a disc is changed by these same operations. We can specify this without reference to the windows on the screen! We therefore construct our new specification as an extension of the M1 spec, rather than the M2 one. The principle that we follow is to use only as much state as is necessary to specify each aspect of product M's behaviour.

We need to talk about the set of all *discs*, and a function mapping each object to the disc on which it resides.

```
home : OBJ —+> DISC
```

It is not always possible to tell what disc an object resides on by looking at the screen , but there is a relationship between the visible hierarchy (loc) and residence on discs: any object which *appears* to be located on a particular disc, is so located. We state this is as

$$\forall d:discs.\ home((loc^{-1})^*\ (\{d\})) = \{d\}$$

Movement and copying is constrained by the discs on which objects reside: specifically, movement never moves an object from one disc to another, while copies are always created on a different disc from their originals.

The full paper defines schemas M3, MOVE3, and COPY3, but space precludes their inclusion at this point.

Note that MOVE3 and COPY3 have mutually exclusive preconditions. Their disjunction, MOVE3 ∨ COPY3, therefore specifies that *either* the object being dragged and the destination reside on different discs, and a copy is performed, *or* they do not, and a move is performed. The choice between moving and copying is determined by the state. This is exactly the effect of dragging one icon onto another, and so we can specify

$$DRAG3 = MOVE3 \vee COPY3$$

(This design decision, that moving and copying be distinguished by the *home* of the objects concerned, can be criticised since the home of an object cannot in general be determined by looking at the screen. The visible hierarchy (loc) does not uniquely determine disc residence (home). In consequence, it is sometimes impossible to determine in advance whether dragging an icon will move it or copy it. Perhaps a design with cut and paste operations for screen objects, consistent with other applications on product A, would have been preferable).

5. The Complete Specification

All that remains is to combine the two extensions we have described into a single, more complex specification. We can do so simply by combining the appropriate schemas.

$$M4 = M2 \wedge M3$$
$$MOVE4 = MOVE2 \wedge MOVE3$$
$$COPY4 = COPY2 \wedge COPY3$$
$$DRAG4 = MOVE4 \vee COPY4$$

6. Conclusions

We have applied the Z specification technique to a visual user interface, product M. Following Sufrin, we adopted an approach of gradually adding detail to a very abstract initial specification. In this way, a moderately complex specification was constructed as a series of extensions, each simple enough to be easily understood. Our specification is by no means a complete description of product M; it could be extended both to describe more operations, and to give much greater detail.

How does product M's design stand up to such close inspection? Pretty well. While constructing the specification I discovered several *regularities* that, as a casual user, I had not previously suspected. Only the strange effects of moving open objects into the trash can, and the occasional impossibility of distinguishing between a move and a copy by looking at the screen, emerged as anomalies.

References

[Hayes] *Specification Case Studies*, ed. I. Hayes, Prentice Hall International, 1987.

[Spivey] *The Z Notation: A Reference Manual*, M. Spivey, Prentice Hall International, 1989.

[Sufrin] *Formal Methods and the Design of Effective User Interfaces*, B. A. Sufrin, in HCI 86, (Cambridge University Press).

SPECIFYING A VISUAL FILE SYSTEM IN Z

John Hughes, Department of Computing Science, University of Glasgow.

1. Introduction

We specify part of the system software of a well-known range of personal computers with a direct manipulation user interface. We refer to this software as product M, and the computers as product A. Product M provides a visual interface to a hierarchical file system. Files, discs, and folders (directories) are visible as icons on the screen. They may be moved or copied from place to place by dragging their icons around with the mouse. Dragging an icon into the "trash can" discards it. Discs and folders may be "opened", creating a window on the screen in which their contents can be viewed. We will specify these aspects of product M, using the Z specification language, and the specification style developed at Oxford.

A system is specified in Z by describing its *states* and the way *operations* affect those states. Both states and operations are specified using *schemas* — collections of variables related by predicates. Schemas are represented by named boxes, divided into an upper part containing variable declarations and a lower part containing predicates involving those variables. Variables after an operation are conventionally given dashed names. To reduce repetition schemas may be used as macros in other schemas, with dashed schemas representing consistent dashing of variables, and ΔS representing both S and S'. If macro expansion gives rise to duplicated declarations, they are considered equivalent to a single declaration.

The specification begins with a very abstract description of product M, to which more detail is added step by step. The Z schema calculus is used to build more detailed specifications from simpler ones, and to combine several detailed views to make the complete specification. This allows the specification of a relatively complex system to be built up from several simple parts, each specifying one aspect of the overall system.

Z is described in [Spivey] and [Hayes], which also contains many case studies of its application. The specification technique used in this paper was inspired by Sufrin's specification of an electronic mail system [Sufrin].

2. Product M's State: A Set of Objects

We focus our attention on the *objects* that the user manipulates directly — icons, possibly with associated windows. Let us introduce a set of all possible objects, containing at least the desktop and the trash can.

$$[OBJ]$$
$$desktop \in OBJ$$
$$trash \in OBJ$$

Every object has a *location*, which is another object. For example, a file might be located in a folder, on a disc, or on the desktop. Locations form a hierarchy. We specify the state as:

M1

$$objects : \mathbb{P} OBJ$$
$$loc : OBJ \pfun OBJ$$

$$desktop \in objects$$
$$trash \in objects$$
$$dom\ loc = objects - \{desktop\}$$
$$rng\ loc \subseteq objects$$
$$loc\ trash = desktop$$
$$(loc^{-1})^* (\{desktop\}) = objects$$

Since $(loc^{-1})^*$ relates each object to everything below it in the hierarchy, $(loc^{-1})^* (\{desktop\})$ is the set of objects reachable from the desktop. So the last line requires that every object is below the desktop in the hierarchy. Coupled with the fact that `loc` is a function, this condition implies that there are no cycles.

2.1 Movement

Now that we can talk about locations, we can begin to specify movement.

MOVE1

$$\Delta M1$$
$$obj? : OBJ \qquad \textit{the object to be moved}$$
$$to? : OBJ \qquad \textit{the destination}$$

$$obj? \notin \{desktop, trash\}$$
$$to? \notin (loc^{-1})^* (\{obj?\})$$
$$loc' = loc \oplus \{obj? \mapsto to?\}$$

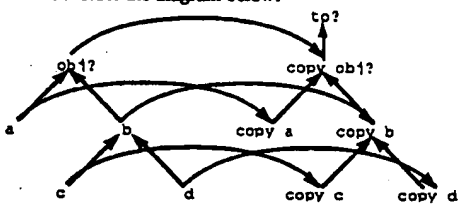
(By convention, the inputs to an operation are given names ending in a question mark.) The third condition above

specifies how the locations of objects are changed, while the second states a precondition that no cycle is created.

2.2 Copying

Product M also provides a way to make a new *copy* of an object, at a new location. The entire hierarchy below the copied object is also copied, so a single operation may create arbitrarily many new objects. We will need to refer to "the copy" of an object, so we introduce a one-to-one function `copy : OBJ >+> OBJ` mapping copied objects to their newly created copies. This association forms part of our description of the COPY operation, so appears in its schema.

The trickiest thing to specify is the copied hierarchy below `copy obj?`, with the same shape as that below `obj?`. How can we express this? Consider the diagram below:



The left side of this diagram represents the hierarchy being copied, the right hand side the new hierarchy, and the curved arrows the `copy` function. But we can pass from `copy d` (say) to its location by: first using $\texttt{copy}^{-1}$ to find d (recall `copy` is one-to-one!), finding the location of d (which is b), and then using `copy` to find the copy of b. So the new hierarchy on the right can be expressed as $\texttt{copy}^{-1} \fatsemi \texttt{loc} \fatsemi \texttt{copy}$ (where $\fatsemi$ denotes function composition).

COPY1

$\Delta M1$

$obj?, to? : OBJ$

$copy : OBJ >+> OBJ$

$obj? \in \{desktop, trash\}$

$dom\ copy = (loc^{-1})^* (\{obj?\})$

$rng\ copy \cap objects = \emptyset$

$loc' = loc \cup \{copy\ obj? \mapsto to?\} \cup copy^{-1} \fatsemi loc \fatsemi copy$

3. Open and Closed Objects

Now let us begin to add more detail, and talk about *open* objects — those having an associated window on the screen displaying their contents, and *visible* objects. We regard an object as visible if it appears in any open window on the screen, including the desktop. (In practice, windows may overlap, and so it may be necessary to rearrange the windows on the screen to see some "visible" objects.)

M2

$M1$

$open, visible : \mathbb{P} OBJ$

$open \subseteq objects$

$desktop \in open$

$visible = loc^{-1}(open)$

It is easy to specify operations to open and close objects.

3.1 Movement

Now let us give a more detailed specification of the MOVE operation.

MOVE2

$\Delta M2$

$MOVE1$

$obj? \in visible$

$to? \in visible \cup open$

$to? \neq trash \Rightarrow open' = open$

$to? = trash \Rightarrow open' = open - \{obj?\}$

The set of open objects is unchanged, *except* when an open object is moved into the trash can. It is then automatically closed. This special behaviour of the trash can seems rather anomolous. There is little wrong with treating the trash can exactly like every other object, and requiring simply `open' = open`. Reasonable alternatives might be to require objects in the trash can to be closed at all times, by modifying the M2 invariant, or to close every object newly below the trash can in the hierarchy, allowing objects to be re-opened once in the trash can. It is hard to understand the rationale for the design actually implemented.

3.2 Copying

In the same way we can add detail to our specification of copying. The open objects change as follows: existing objects remain open if they were open before the operation, while copies of open objects are created open. The set of newly open objects is therefore the image of the set of open objects through the `copy` function.

```
COPY2 ──────────────────────────────
|     ΔM2
|     COPY1
|-----------------
|     obj? ∈ visible
|     to? ∈ visible ∪ open
|     open' = open ∪ copy (open)
──────────────────────────────────
```

4. Residence on Discs

Our second extension specifies explicitly on which disc objects reside, and how residence on a disc is changed by these same operations. We can specify this without reference to the windows on the screen! We therefore construct our new specification as an extension of the M1 spec, rather than the M2 one. The principle that we follow is to use only as much state as is necessary to specify each aspect of product M's behaviour.

We need to talk about the set of all *discs*, and a function mapping each object to the disc on which it resides.

```
home : OBJ —+> DISC
```

It is not always possible to tell what disc an object resides on by looking at the screen , but there is a relationship between the visible hierarchy (`loc`) and residence on discs: any object which *appears* to be located on a particular disc, is so located. We state this is as

$$\forall d:discs.\ home((loc^{-1})^*\ \{\{d\}\}) = \{d\}$$

Movement and copying is constrained by the discs on which objects reside: specifically, movement never moves an object from one disc to another, while copies are always created on a different disc from their originals.

The full paper defines schemas M3, MOVE3, and COPY3, but space precludes their inclusion at this point.

Note that MOVE3 and COPY3 have mutually exclusive preconditions. Their disjunction, $MOVE3 \vee COPY3$, therefore specifies that *either* the object being dragged and the destination reside on different discs, and a copy is performed, *or* they do not, and a move is performed. The choice between moving and copying is determined by the state. This is exactly the effect of dragging one icon onto another, and so we can specify

$$DRAG3 = MOVE3 \vee COPY3$$

(This design decision, that moving and copying be distinguished by the *home* of the objects concerned, can be criticised since the home of an object cannot in general be determined by looking at the screen. The visible hierarchy (`loc`) does not uniquely determine disc residence (`home`). In consequence, it is sometimes impossible to determine in advance whether dragging an icon will move it or copy it. Perhaps a design with cut and paste operations for screen objects, consistent with other applications on product A, would have been preferable).

5. The Complete Specification

All that remains is to combine the two extensions we have described into a single, more complex specification. We can do so simply by combining the appropriate schemas.

$$M4 = M2 \wedge M3$$
$$MOVE4 = MOVE2 \wedge MOVE3$$
$$COPY4 = COPY2 \wedge COPY3$$
$$DRAG4 = MOVE4 \vee COPY4$$

6. Conclusions

We have applied the Z specification technique to a visual user interface, product M. Following Sufrin, we adopted an approach of gradually adding detail to a very abstract initial specification. In this way, a moderately complex specification was constructed as a series of extensions, each simple enough to be easily understood. Our specification is by no means a complete description of product M; it could be extended both to describe more operations, and to give much greater detail.

How does product M's design stand up to such close inspection? Pretty well. While constructing the specification I discovered several *regularities* that, as a casual user, I had not previously suspected. Only the strange effects of moving open objects into the trash can, and the occasional impossibility of distinguishing between a move and a copy by looking at the screen, emerged as anomalies.

References

[Hayes] *Specification Case Studies*, ed. I. Hayes, Prentice Hall International, 1987.

[Spivey] *The Z Notation: A Reference Manual*, M. Spivey, Prentice Hall International, 1989.

[Sufrin] *Formal Methods and the Design of Effective User Interfaces*, B. A. Sufrin, in HCI 86, (Cambridge University Press).